Chapter 3
Processes

Da-Wei Chang

CSIE.NCKU

Source: Abraham Silberschatz, Peter B. Galvin, and Greg Gagne, "Operating System Concepts", 10th Edition, Wiley.

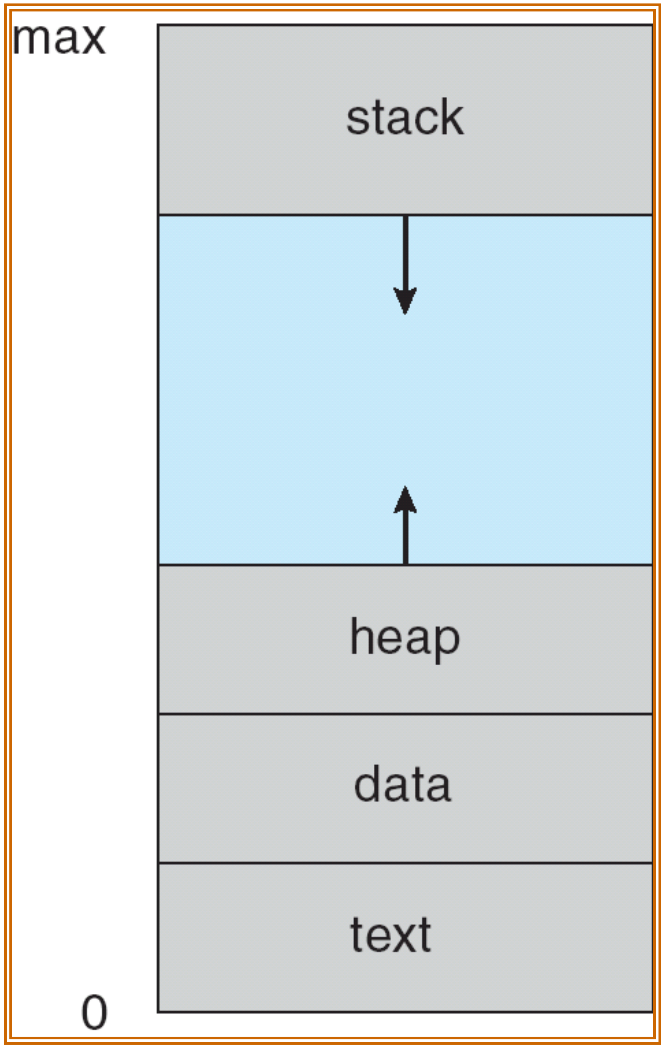
Outline

- Process Concept
- Process Scheduling
- Operations on Processes
- Inter-Process Communication (IPC)
- Examples of IPC Systems
- Communication in Client-Server Systems

Process Concept

- An operating system executes user programs
  - Batch system – jobs
  - Time-sharing systems – processes or tasks
- We use the terms *job* and *process* almost interchangeably
- Process – a program in execution; process execution must progress in sequential fashion
- A process includes
  - text (i.e., code section)
  - data section
  - heap
  - stack
  - program counter and the content of the processor registers
- Program – ***passive*** ; Process – ***active***

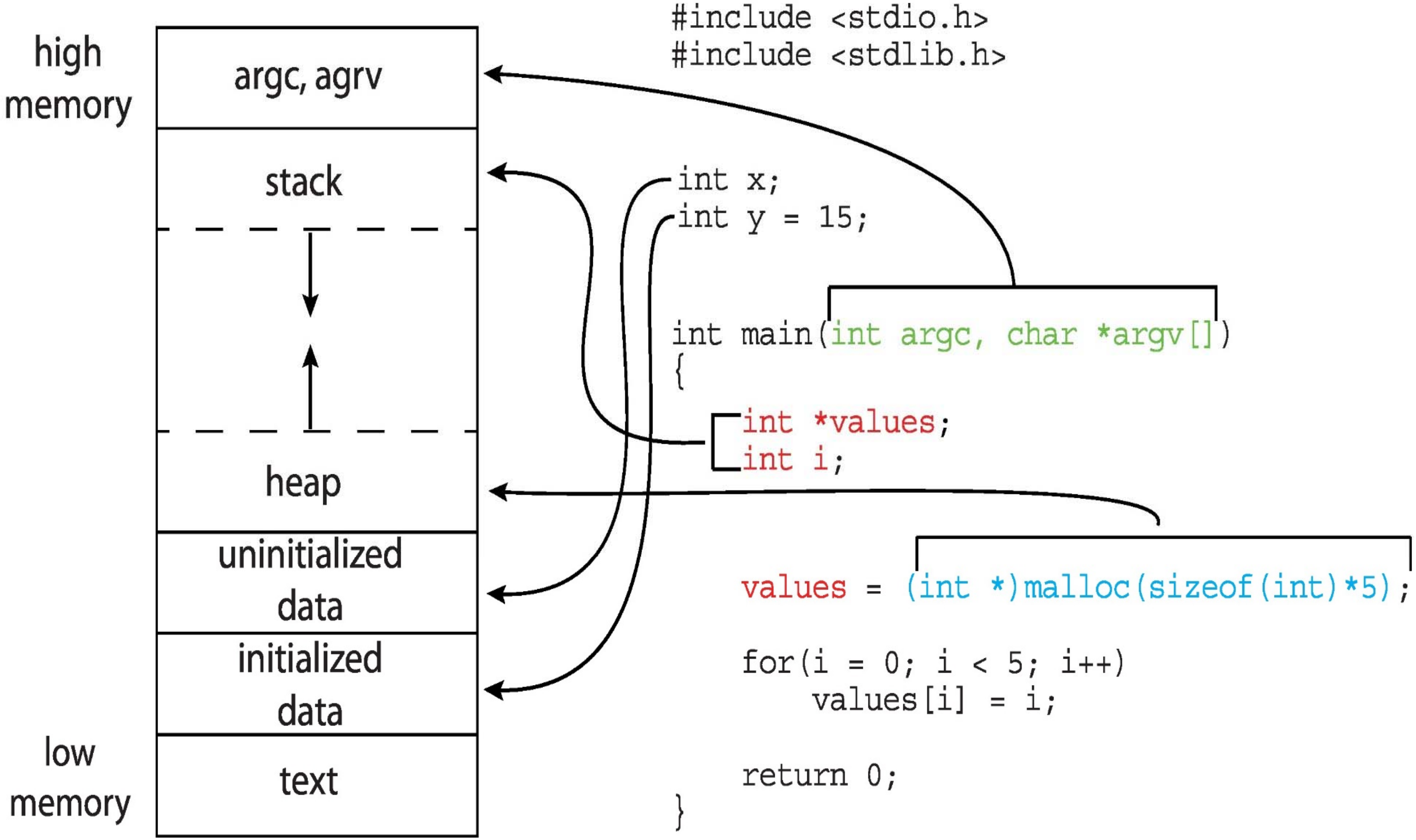
A Process in Memory



More than one processes can be associated with the same program

Which parts are the same for these processes?

Memory Layout of a C Program

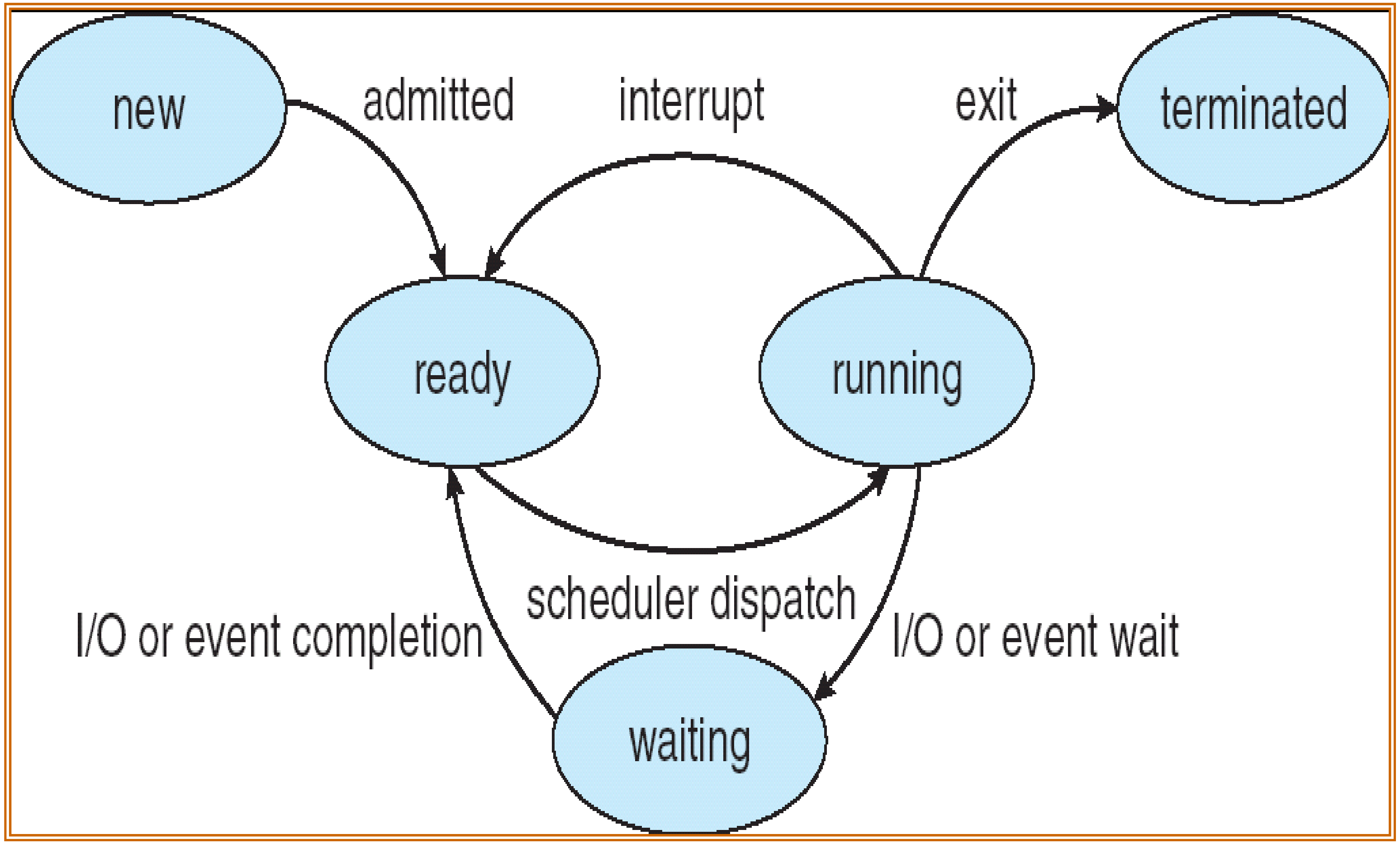


```
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 15;


int main(int argc, char *argv[])
{
   int *values;
   int i;


   values = (int *)malloc(sizeof(int)*5);

   for(i = 0; i < 5; i++)
      values[i] = i;

   return 0;
}
```

Process State

- As a process executes, it changes its *state*
  - **new**: The process is being created
  - **ready**: The process is waiting to be assigned to a CPU
    - The process is **runnable**
  - **running**: Instructions are being executed **(i.e., owns the CPU)**
  - **waiting**: The process is waiting for some event to occur
  - **terminated**: The process has finished execution

Diagram of Process State



Each process has its own process state diagram!!!

Process Control Block (PCB)

Information associated with each process

- Process state
- Program counter
  - The address of the ***next*** instruction
  - Must be saved when an interrupt occurs
- CPU registers
  - Vary in number and type, depending on the processor architecture
    - Accumulators, index registers, stack pointers, general purpose registers…
  - Must be saved when an interrupt occurs
- CPU scheduling information
  - Priority, pointers to scheduling queues, other scheduling parameters…

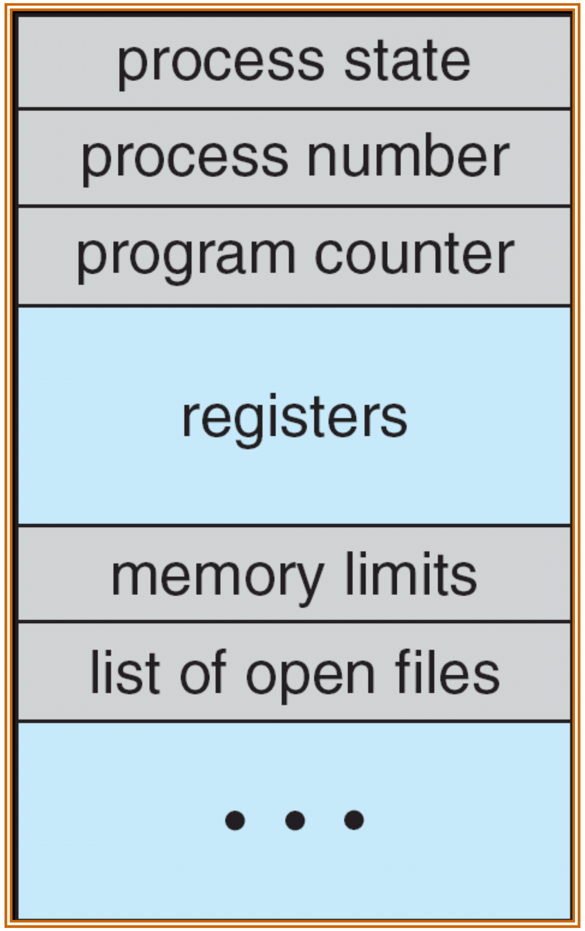
Process Control Block (PCB)

Information associated with each process (cont.)

- Memory-management information
  - Will be introduced later
- Accounting & identification information
  - CPU time and real time used
  - Time limits
  - Process numbers
- I/O status information
  - Open files and devices…

Process Control Block (PCB)

Also called Task Control Block (TCB)



process state

process number

program counter

registers

memory limits

list of open files

• • •

task_struct in Linux: /include/linux/sched.h

Threads

- So far, a process is a program that performs a **single thread** of execution
  - Process executes in sequential fashion
  - Many modern operating systems support multi-threaded processes
  - Discussed in Chapter 4

Process Scheduling

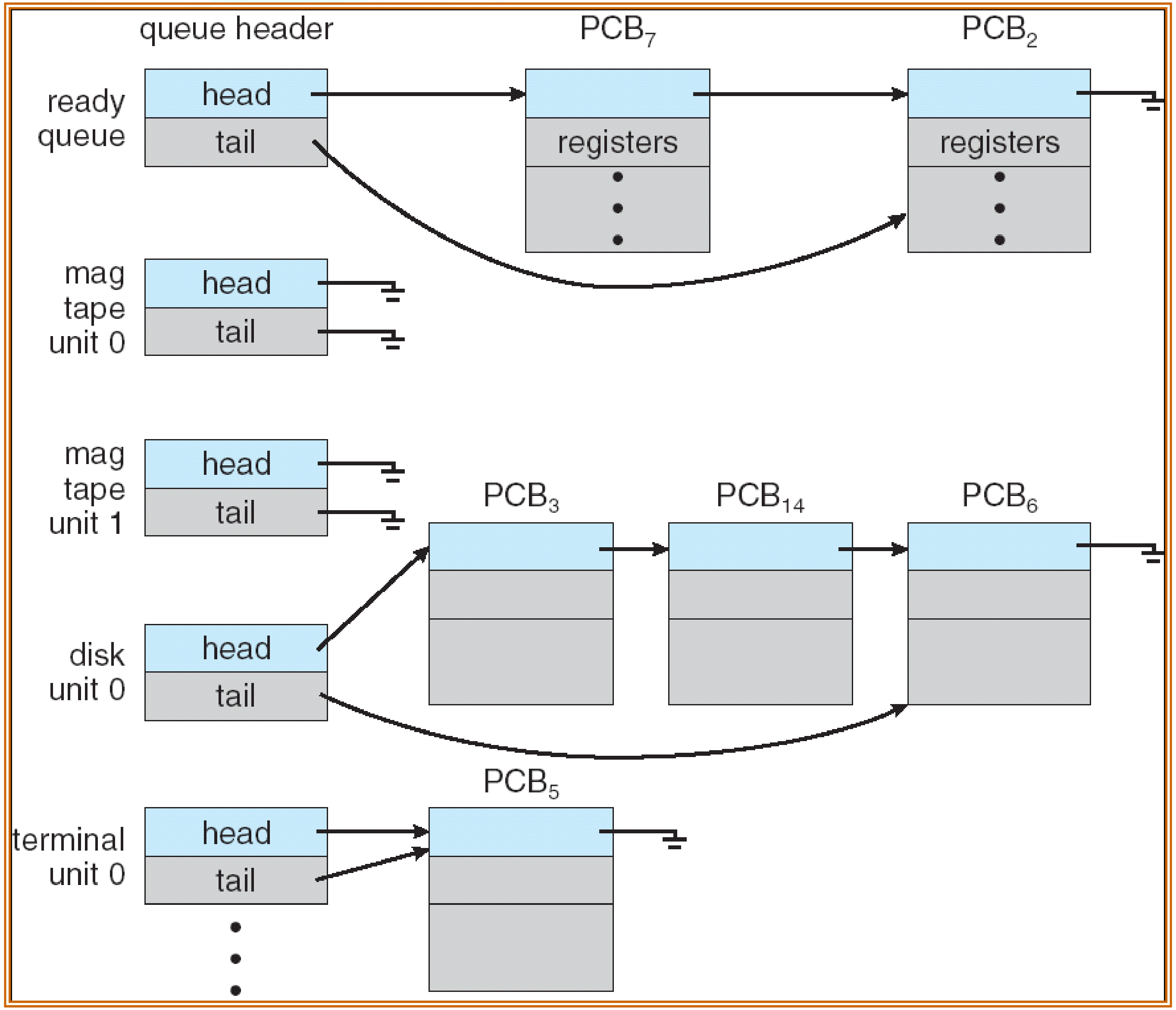
- Goals of
  - Multiprogramming
    - High CPU utilization
  - Time-sharing
    - Short response time

- A process scheduler is responsible for selecting a process to run, trying to achieve the above goals

Process Scheduling Queues

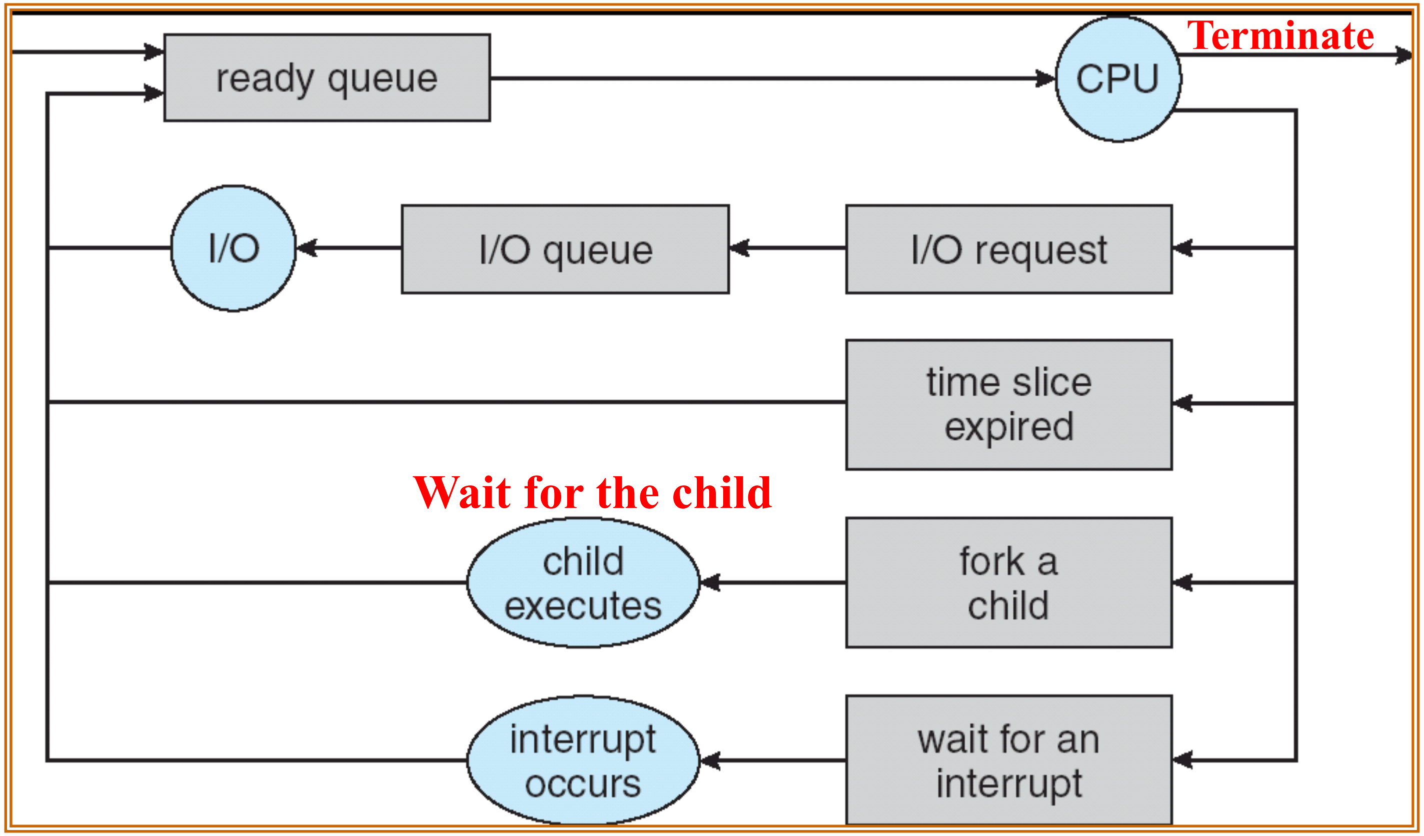
- **Job queue** – set of all processes in the system
- **Ready queue** (or called run queue) – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device
- **Event queues** – set of processes waiting for an event
- **Processes migrate among the queues**

Ready Queue and Various I/O Device Queues



Queuing-Diagram Representation of Process Scheduling

Create

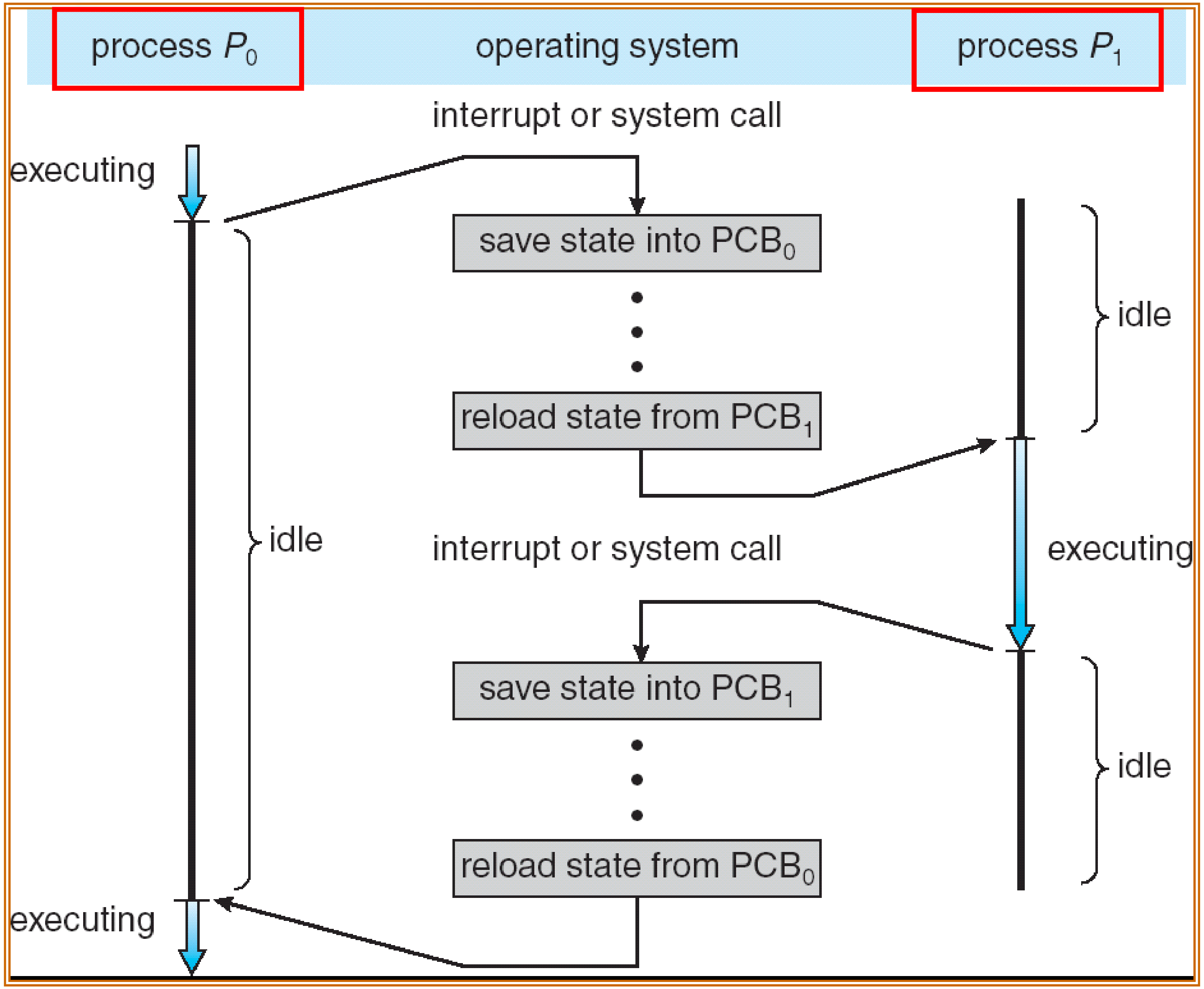


Free PCB and resources

Context Switch

- When CPU switches to another process, the system must save the state of the old process in its PCB and load the saved state for the new process
  - This is called context switch
- Context-switch time is overhead; the system does no useful work while switching
- Time dependent on hardware support
  - e.g., SUN UltraSPARC provide multiple set of registers
    - Context switch == change the pointer to the current register set
  - Typically, context switch requires a few microseconds

CPU Switch from Process to Process



Schedulers

- **Long-term scheduler** (or Job scheduler) – determines which processes should be brought into the ready queue from the job queue

- **Short-term scheduler** (or CPU scheduler) – determines which process in the ready queue should be executed next (on the CPU)

Schedulers (Cont.)

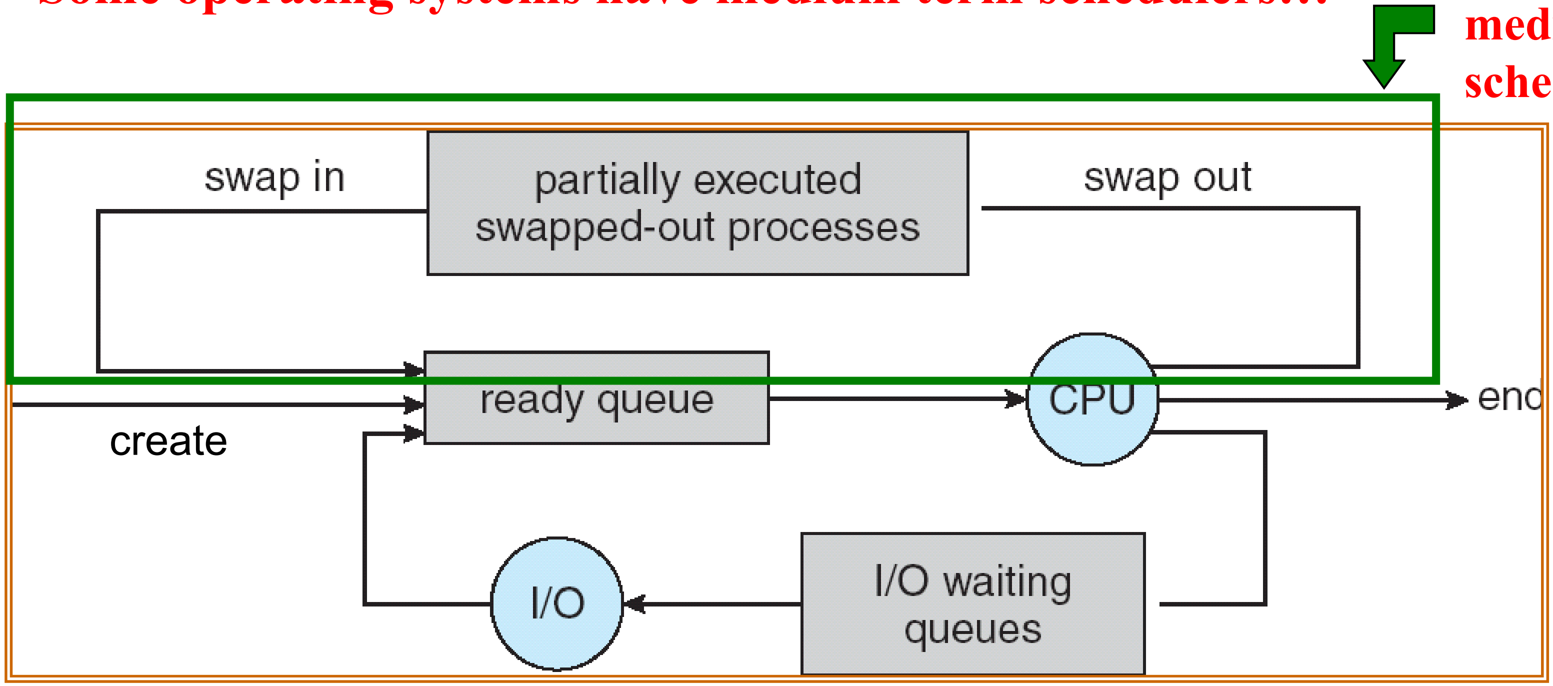
- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast)
  - – E.g., 100 ms for each process, 10ms for performing scheduling
    - 10 / (100+10) = 9% CPU time wasted on the scheduling
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (can be slow)

Schedulers (Cont.)

- The long-term scheduler controls the *degree of multiprogramming*
- Processes can be described as either
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- A long-term scheduler should select a good process mix
  - All IO-bound? All CPU-bound?
- Windows and UNIX have no long-term schedulers
  - Put all the jobs into memory

Addition of Medium Term Scheduling

Some operating systems have medium term schedulers…



Needed

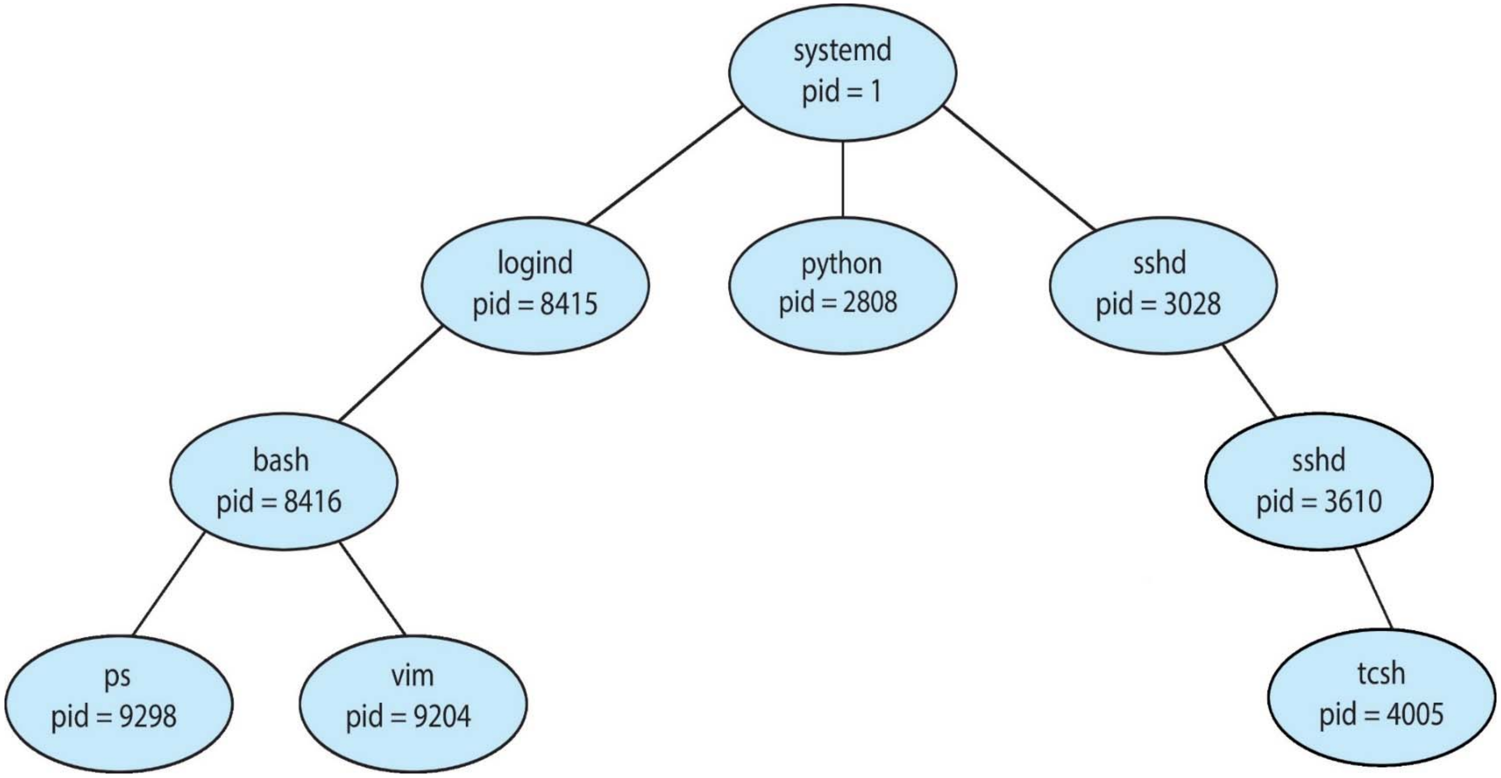
- **for improving the process mix**
- **when a change of memory requirement has overcommitted the available memory**

Operations on Processes

- We introduce two operations here
  - Creation
  - Termination

Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes



Generally, a process identified via a **process identifier** (**pid**)

Process Creation (Cont.)

- Parent and Child
  - Resource allocation
    - A child can obtain its resources (CPU time, mem, files, IO dev…) from
      a) OS
      b) Its parent
    - If a child can **only** obtain its resources from its parent
      - Prevent any process from overloading the system by creating too many processes

Process Creation (Cont.)

- Parent and Child
  - Resource sharing
    - Parent and child share all resources
    - Child shares subset of parent’s resources
    - Parent and child share no resources
  - Execution
    - Parent and child execute concurrently, or
    - Parent waits until child terminates
  - Address space
    - Child runs in the same address space as its parent, or
    - Child has its own address space

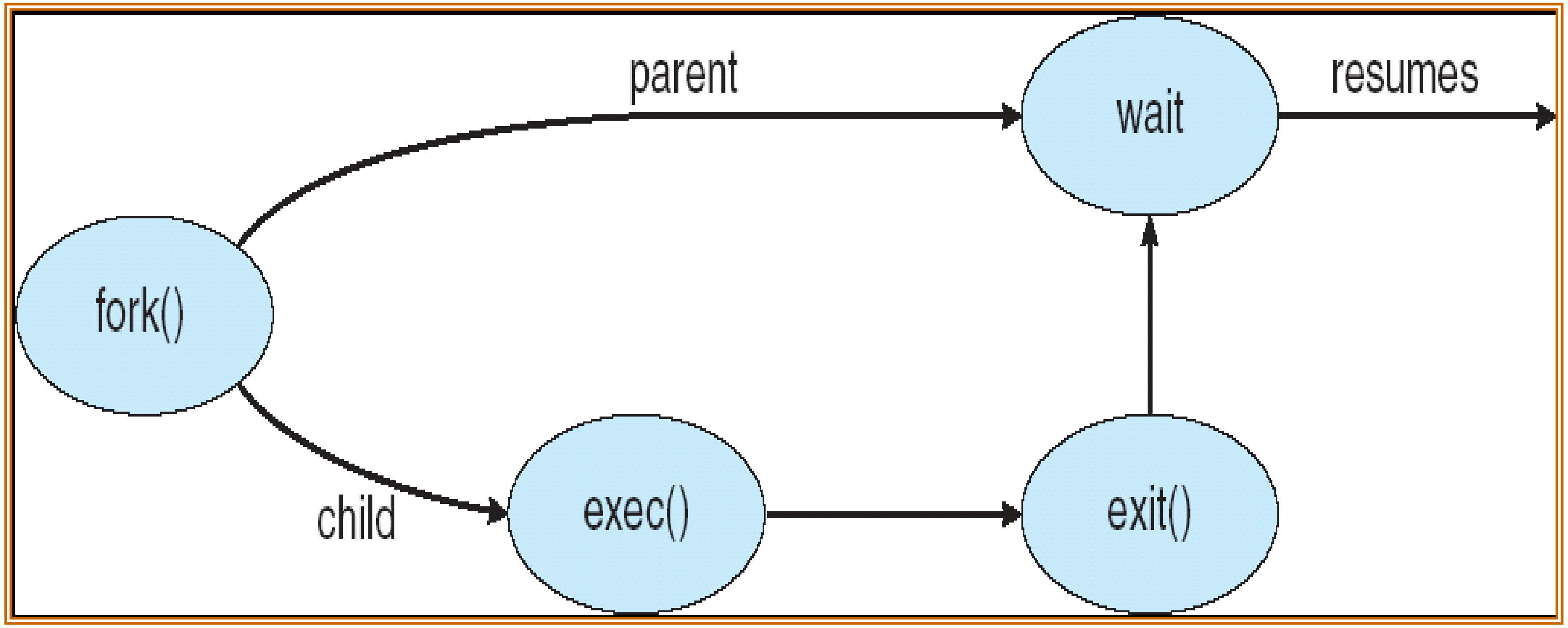
Process Creation (Cont.)

- UNIX examples
  - **fork** system call creates new process
  - **exec** system call
    - **replace** the process' memory space with a new program
    - usually used after a **fork**

C Program Forking Another Process

```
int main()
{
    pid_t ret, dead;
    int status;

    ret = fork();       /* fork another process */
    if (ret < 0) {      /* error occurred */
            fprintf(stderr, "Fork Failed");
            exit(-1);
    }
    else if (ret == 0) { /* child process */
            execlp("/bin/ls", "ls", NULL);
    }
    else {  /* parent process */   // ret == child's pid
            /* parent will wait for the child to complete */
            dead = wait (&status); // dead == the pid of the child that has died
            printf ("Child Complete");
            exit(0);
    }
}
```

Execution Flow of the Program



Process Termination

- Process executes last statement and asks the operating system to delete it (via the **exit()** system call)
  - Output data/status to its parent
    - The parent can get the exit status via the **wait()** system call
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes
  - Reasons
    - Child has exceeded allocated resources
    - Task assigned to child is no longer required
- If parent is exiting
  - Some operating system (such as VMS) do not allow child to continue if its parent terminates
    - All children terminated - *cascading termination*
  - In UNIX, cascading termination is not required
    - Child's parent is set to the init/systemd process (pid = 1)

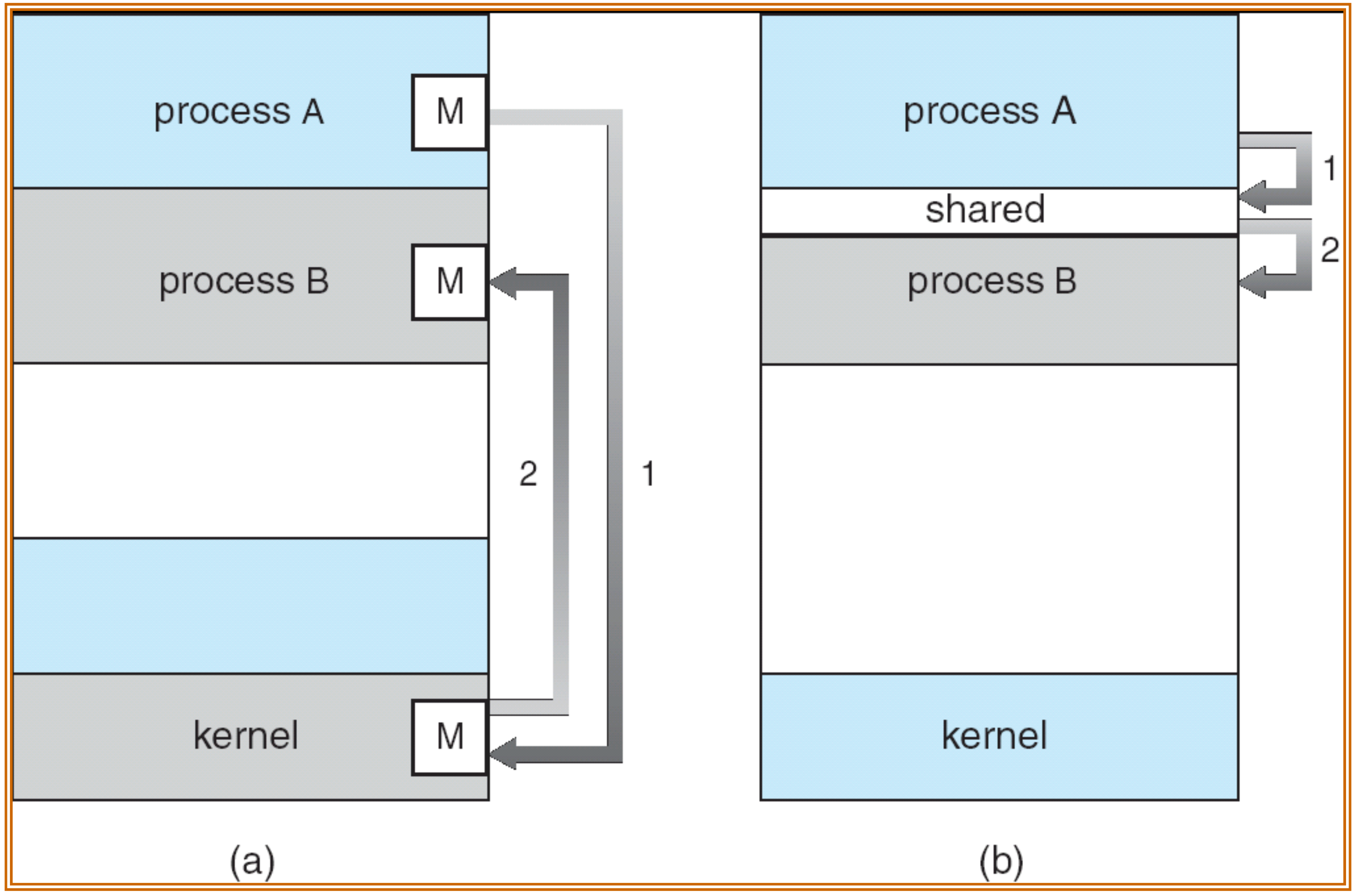
Inter-Process Communication (IPC)

- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
  - Any processes that share data with others are cooperating processes
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
    - Divide the jobs and run them in parallel on a MP system
  - Modularity

Inter-Process Communication (IPC)

- IPC allows processes to exchange data
- Two fundamental models
  - Shared memory
    - Faster
      - Requires **no** system calls when reading/writing data
  - Message passing
    - Useful for exchanging smaller data
    - Easier to implement
    - Slower; requires system calls for sending/receiving messages

Communications Models



Message passing **Shared memory**

Shared Memory

- A shared memory region must be created first
- Then, it is attached to the address spaces of the processes
- Memory access can be done freely in the region
  - Originally, OS prevents the address space of a process to be accessed by the other processes.
  - However, the users/processes of the shared memory remove this restriction
  - OS do not care about the data written in the shared memory region
- The region can be detached if it is no longer needed by a process

Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information (i.e. data items) that is consumed by a *consumer* process
  - Use a buffer to contain the data items
  - ***unbounded-buffer*** places no practical limit on the size of the buffer
    - Producer never has to wait; consumer may need to wait
  - ***bounded-buffer*** assumes that there is a fixed buffer size
    - Both producer and consumer may need to wait

Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
// in == out : empty,  (in+1) mod BUFFER_SIZE == out: full
```

- **Solution is correct, but can only use BUFFER_SIZE-1 elements**

Bounded-Buffer – Producer

```
while (true)
{
  …Produce an item…
   while ( ((in + 1) % BUFFER SIZE)  == out )
       ;   /* do nothing -- no free buffers */
   buffer[in] = item;
   in = (in + 1) % BUFFER SIZE;
}
```

Bounded Buffer – Consumer

```
while (true)
{
          while (in == out)
                    ; // do nothing -- nothing to consume

         // remove an item from the buffer
         item = buffer[out];
         out = (out + 1) % BUFFER SIZE;
         …Process the item…
}
```

Shared Memory APIs

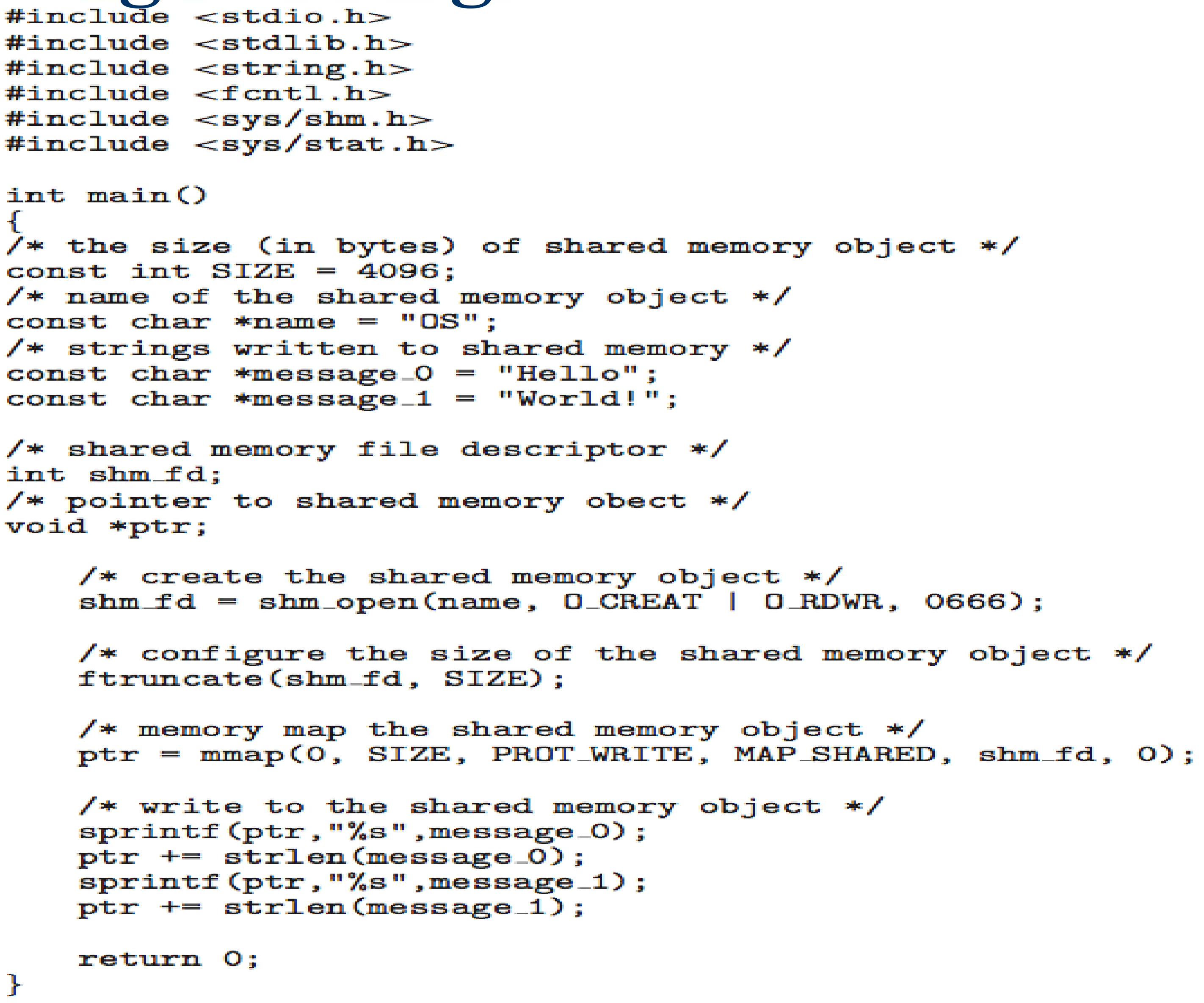
- System V API
  - shmget()
  - shmat()
  - shmdt()
  - shmctl()
- POSIX (Portable Operating System Interface) Shared Memory API
  - shm_open(), mmap(), munmap(), close(), shm_unlink()…

POSIX Shared Memory Example

- POSIX Shared Memory
  - Process first creates shared memory segment
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
    - Also used to open an existing segment
  - Set the size of the object

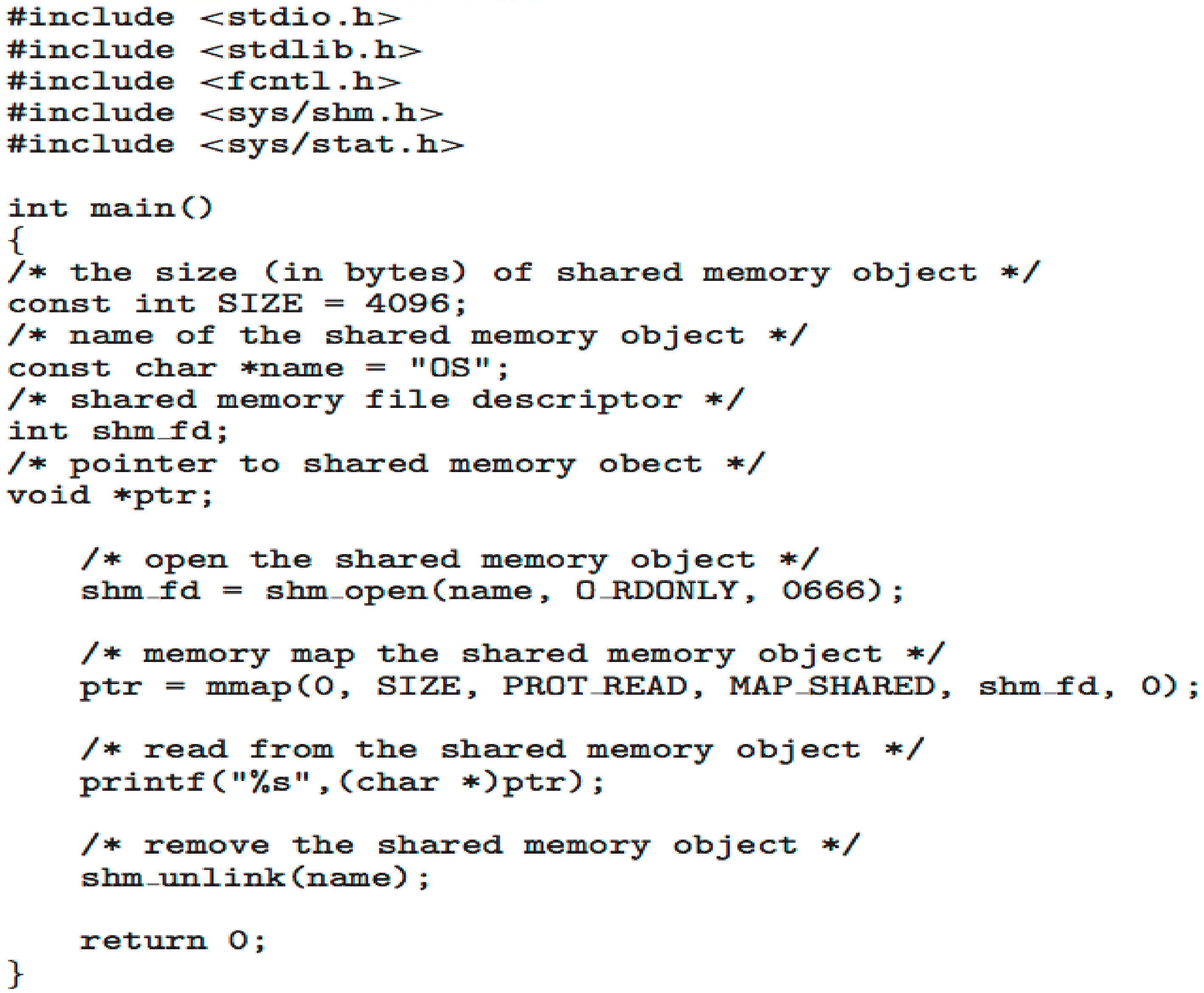
    ftruncate(shm_fd, 4096);
  - Use **mmap()** to memory-map the shared memory object to the address space
  - Reading and writing to shared memory is done by using the pointer returned by mmap()

POSIX Shared Memory Example – Writing a String



```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

    /* write to the shared memory object */
    sprintf(ptr,"%s",message_0);
    ptr += strlen(message_0);
    sprintf(ptr,"%s",message_1);
    ptr += strlen(message_1);

    return 0;
}
```

POSIX Shared Memory Example – Reading a String



```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* open the shared memory object */
   shm_fd = shm_open(name, O_RDONLY, 0666);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

   /* read from the shared memory object */
   printf("%s",(char *)ptr);

   /* remove the shared memory object */
   shm_unlink(name);

   return 0;
}
```

Message Passing

- Message system – processes communicate with each other without resorting to shared variables/memory
- Two major operations
  - **send**(*message*)
  - **receive**(*message*)
- Message size can be fixed or variable
- If *P* and *Q* wish to communicate, they need to
  - have a *communication link* between them
  - exchange messages via send/receive

Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message fixed or variable?
- Is a link unidirectional or bi-directional?

Direct Communication

- Processes must name each other explicitly:
  - **send** (*P*, *message*) – send a message to process P
  - **receive**(*Q*, *message*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - Need to know each other's identity
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
- The API above is symmetric in addressing
- Asymmetric addressing
  - **send** (*P*, *message*) – send a message to process P
  - **receive**(*&id* , *message*) – receive a message, the id returns the sender
- Drawback
  - Hard-coding process id
    - Changing process id would cause problems…

Indirect Communication

- Messages are sent to and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Indirect Communication

- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:

  send(*A*, *message*) – send a message to mailbox A

  receive(*A*, *message*) – receive a message from mailbox A

Mailbox Ownership

- A mailbox is owned by the creator or the OS
  - Mailboxes owned by the Creator
    - Mailbox is part of the address space of the creator
    - The creator is the only receiver
    - Creator terminates → mailbox disappears…
  - Mailboxes owned by the OS
    - May be still existed after the creator has terminated
    - OS should provide a mechanism for users to delete the mailbox explicitly

Indirect Communication

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1 sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** has the sender send the message and continue ➔ does not wait for the receiver
  - **Non-blocking receive** has the receiver receive a valid message or null

Message Buffering

- A queue of messages attached to the link
- Implemented in one of three ways
  - Zero capacity – 0 message
    Sender must wait for receiver
    - No buffering
  - Bounded capacity – finite length of *n* messages
    Sender must wait if link full
    - Automatic buffering
  - Unbounded capacity – infinite length
    Sender never waits
    - Automatic buffering
- Receivers have to wait if there is no message (in the case of blocking receive)

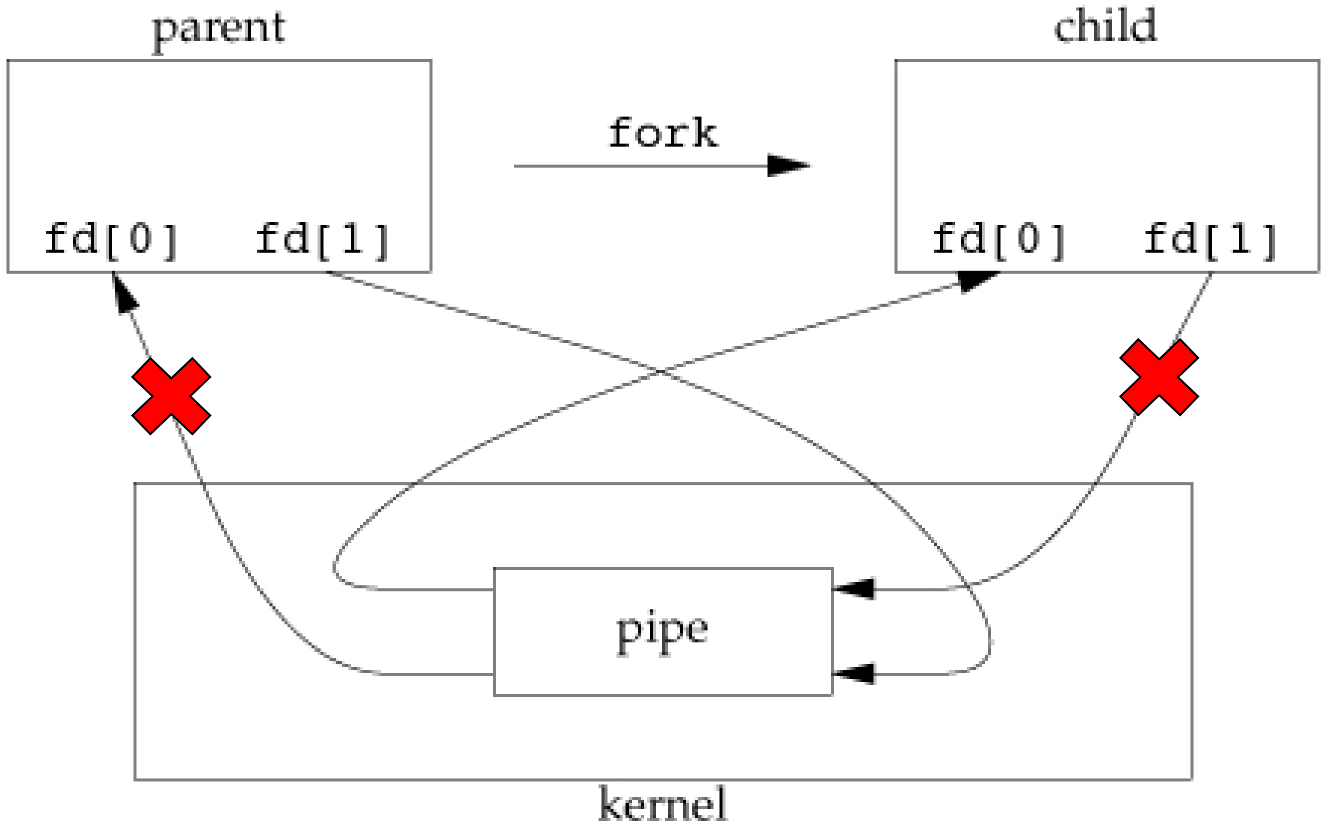
Message Passing APIs

- System V API
  - msgget()
  - msgsnd()
  - msgrcv()
  - msgctl()
- POSIX Message Queue API
  - mq_xxx() functions
    - mq_open(), mq_send(), mq_receive(), mq_close(), mq_unlink()….

Pipes

- A container for byte stream
- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Pipes are typically unidirectional
  - According to POSIX.1-2001
  - On some systems, pipes are bidirectional
- Require parent-child relationship between communicating processes
- Windows calls these **anonymous pipes**

IPC Using Pipes



Source: http://www.cs.columbia.edu/~jae/4118/L05-ipc.html

IPC Using Pipes – An Example

```
…
int main(int argc, char *argv[])
{
    int pipefd[2];
    pid_t cpid;
    char buf;
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <string>\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
    cpid = fork();
    if (cpid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
```

IPC Using Pipes – An Example

```
    if (cpid == 0) {    /* Child reads from pipe */
        close(pipefd[1]);          /* Close unused write end */
        while (read(pipefd[0], &buf, 1) > 0)
            write(STDOUT_FILENO, &buf, 1);
        write(STDOUT_FILENO, "\n", 1);
        close(pipefd[0]);
        exit(EXIT_SUCCESS);

    } else {            /* Parent writes argv[1] to pipe */
        close(pipefd[0]);          /* Close unused read end */
        write(pipefd[1], argv[1], strlen(argv[1]));
        close(pipefd[1]);          /* Reader will see EOF */
        wait(NULL);                /* Wait for child */
        exit(EXIT_SUCCESS);
    }

  }
```

Named Pipes

- Also called FIFOs
  - On Linux, name pipes are created by **mkfifo**()
- Named pipes are more powerful than ordinary pipes
  - No parent-child relationship is necessary between the communicating processes
  - Any process can open the FIFO for reading or writing since it has a name
- Provided on both UNIX and Windows systems

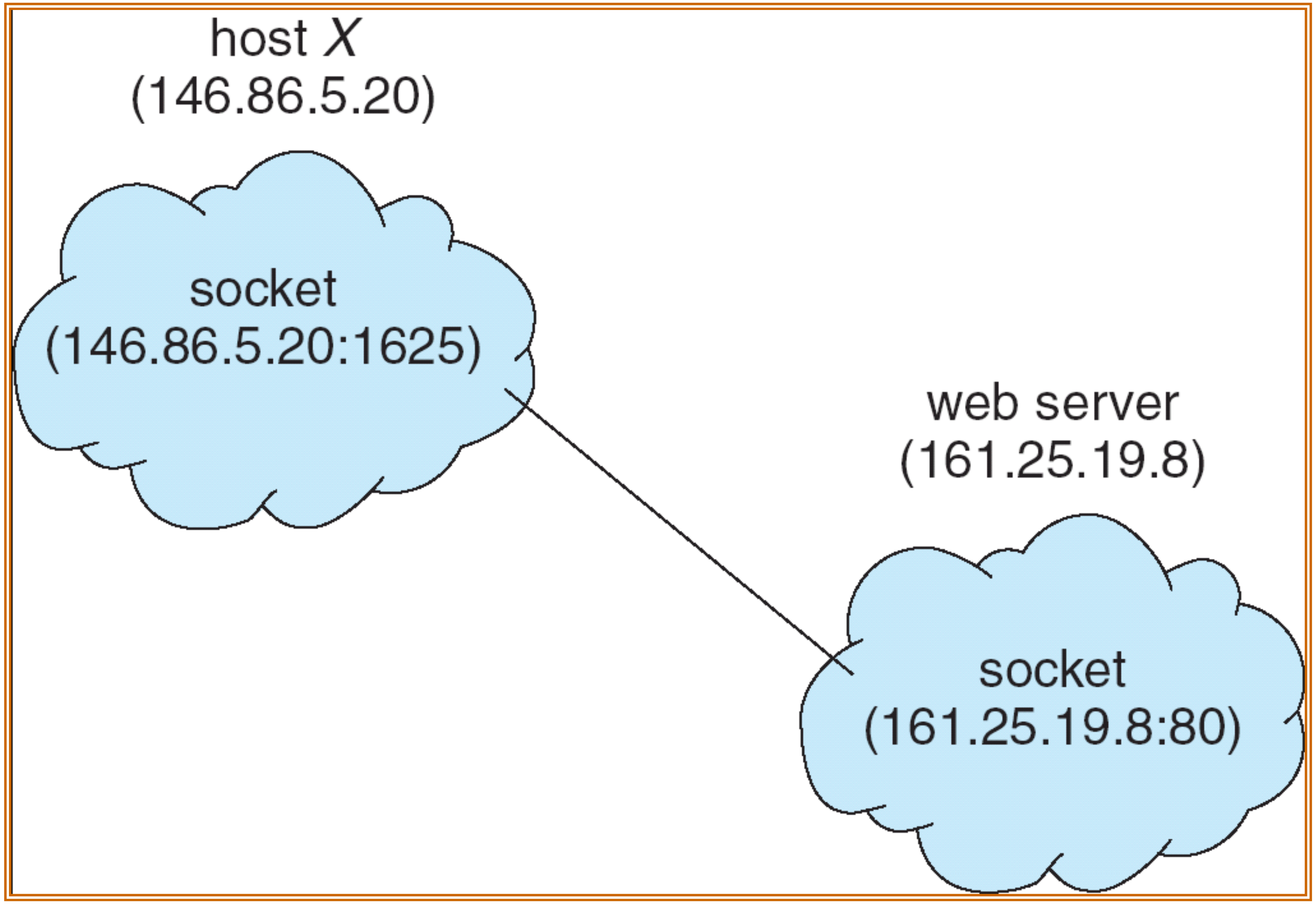
Client-Server Communication

- Sockets
- Remote Procedure Calls
- Remote Method Invocation (Java)

Sockets

- A socket is defined as an *endpoint for communication*
- Concatenation of IP address and port
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
  - Some well known ports
    - FTP server: 21
    - HTTP server: 80
- Communication consists between a pair of sockets

Socket Communication



Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between **processes on networked systems**
- Allow a client to **invoke a procedure** ***on a remote host*** as it would invoke a procedure locally
- Based on message passing
  - Each message is sent to the RPC daemon listening to a port
    - A message contains function id, and function parameters
  - Results are sent back as separated messages

Remote Procedure Calls

- **Stubs**
  - transforms procedure calls to msgs, and vice versa.
  - Hide the communication details
- The client-side stub locates the server and *marshals* the parameters
- The server-side stub receives this message, unpacks the marshaled parameters, and performs the procedure on the server
- The return values are processed in a similar manner

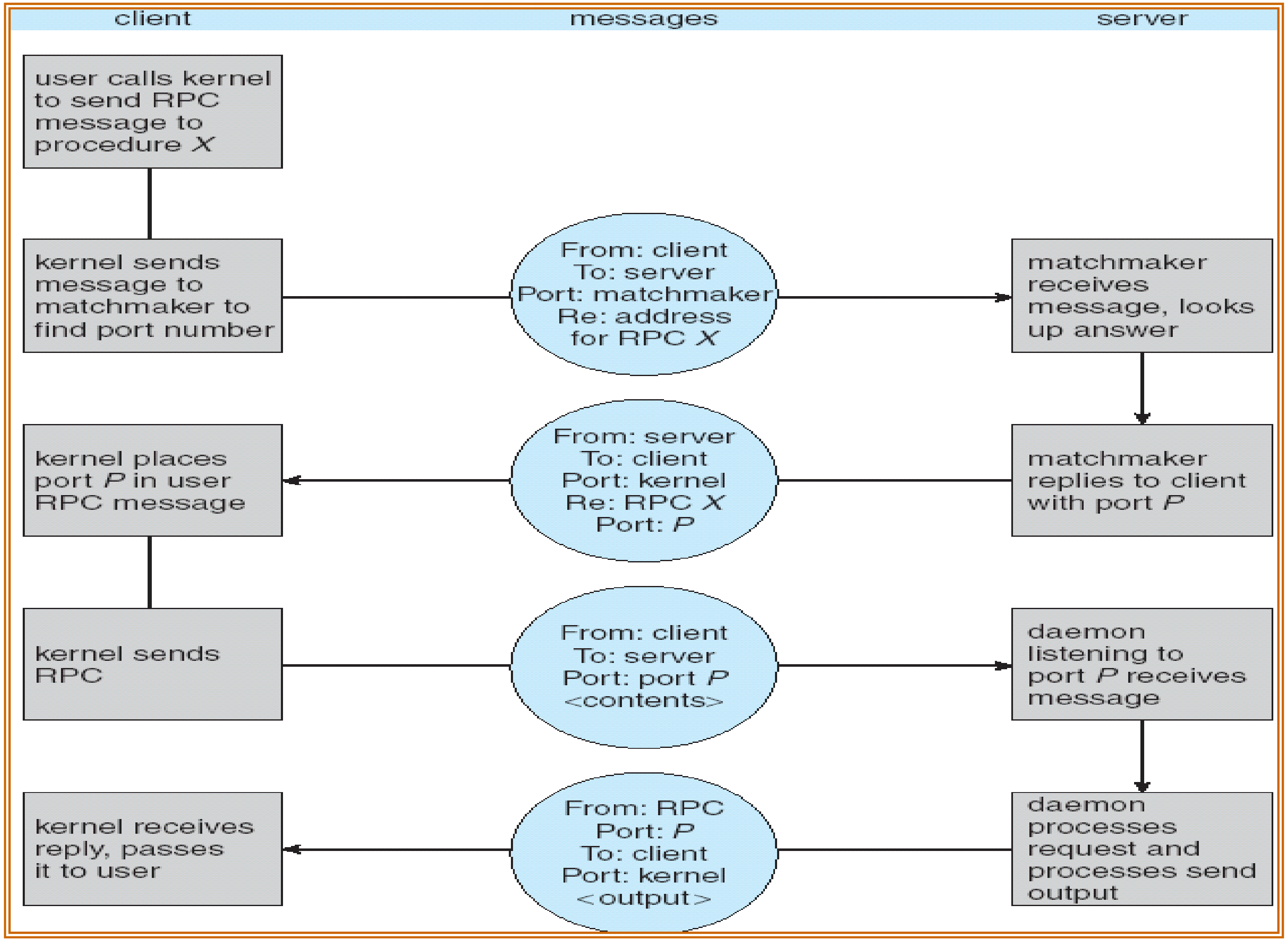
Issues about RPC

- Data representation
  - Little Endian? Big Endian?
  - External Data Representation (XDR)
- Error handling
  - RPC can fail, or be executed more than once due to network error
  - OS ensures the “exactly once” semantic by (ALO && AMO)
    - At least once (ALO)
      - Server acks the client when the client msg has been received & executed
      - Client msg transmission repeats until ACKed
    - At most once (AMO)
      - Each msg has a timestamp
      - Drop the msg if its timestamp is in the *history* on the server

Issues about RPC

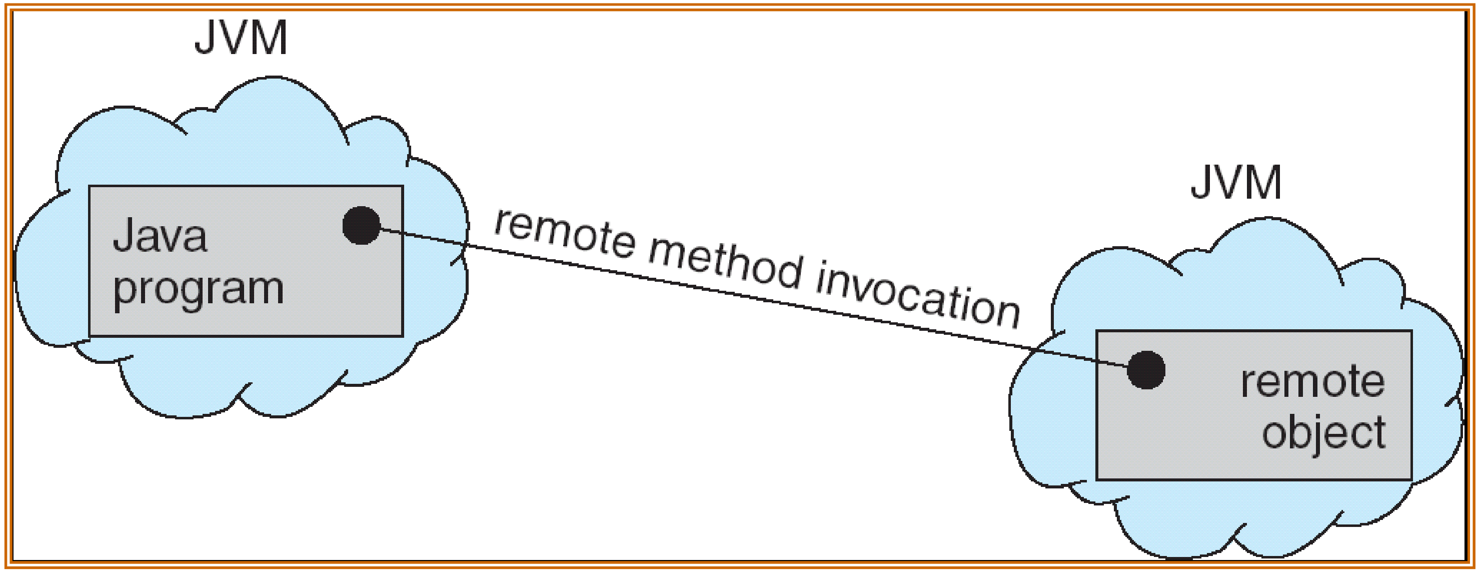
- Binding
  - How does the client knows the port number of the server?
    - Fixed port number
      - Inflexible
        - » Server can not change its port easily
    - Dynamic binding
      - A rendezvous daemon (a matchmaker) listening on a fixed port to find out the port of the server
      - See the next slide

Execution of RPC

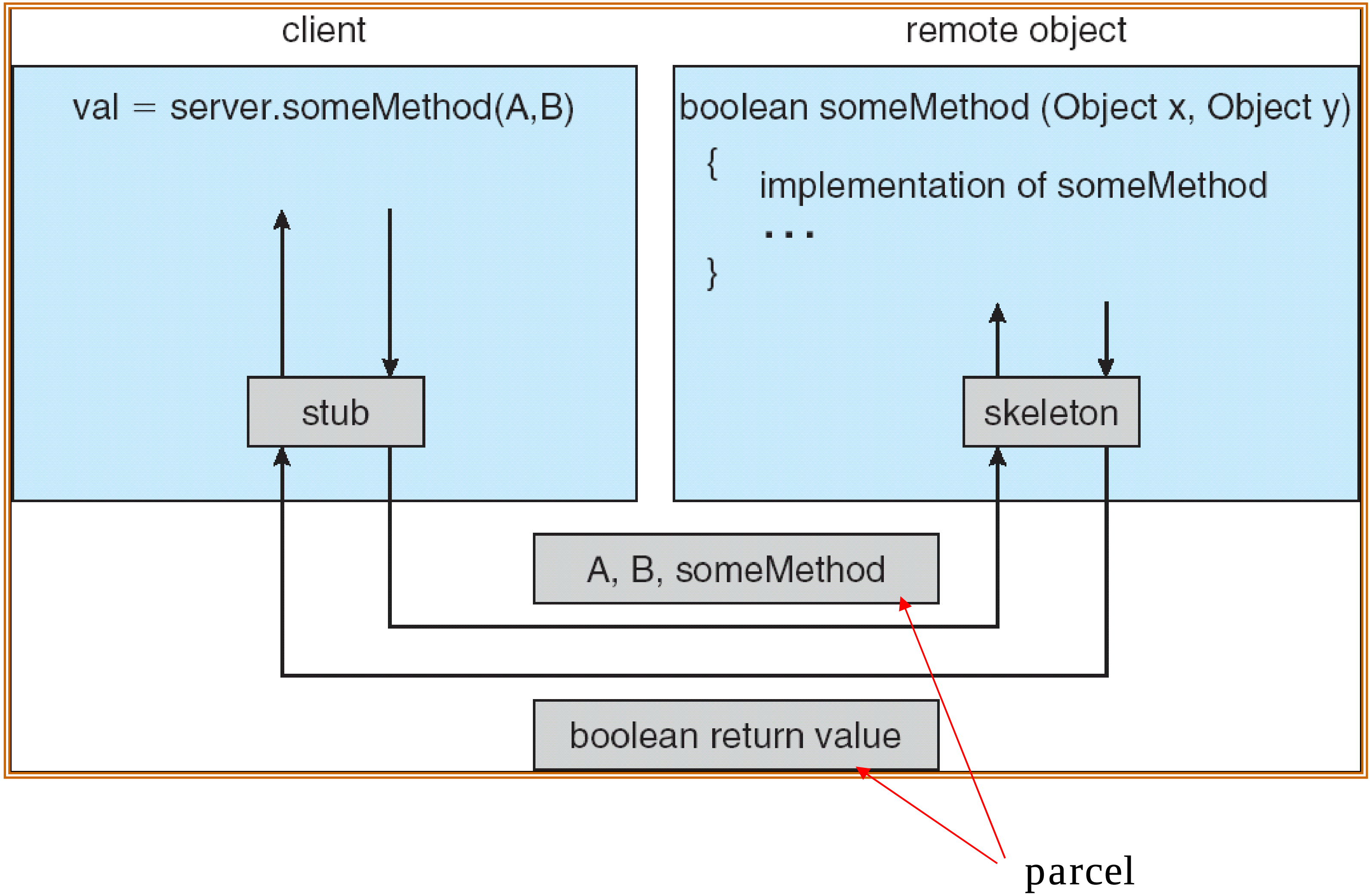

client
messages
server
user calls kernel to send RPC message to procedure X
kernel sends message to matchmaker to find port number
From: client To: server Port: matchmaker Re: address for RPC X
matchmaker receives message, looks up answer
matchmaker replies to client with port P
From: server To: client Port: kernel Re: RPC X Port: P
kernel places port P in user RPC message
kernel sends RPC
From: client To: server Port: port P <contents>
daemon listening to port P receives message
daemon processes request and processes send output
From: RPC Port: P To: client Port: kernel <output>
kernel receives reply, passes it to user

Remote Method Invocation

- Remote Method Invocation (RMI) is a Java mechanism similar to RPCs
- RMI allows a Java program on one machine to invoke a method on a remote object
- Different from RPC, RMI is
  - Object-based
  - Possible to pass objects as parameters



Marshalling Parameters


client
remote object
val = server.someMethod(A,B)
boolean someMethod (Object x, Object y)
{
implementation of someMethod
. . .
}
stub
skeleton
A, B, someMethod
boolean return value
parcel